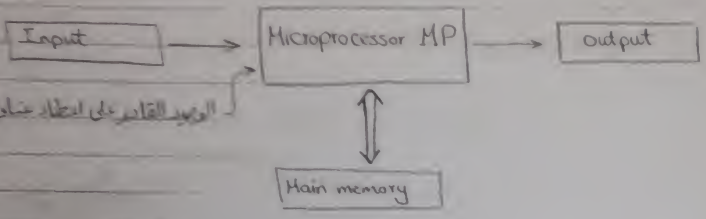
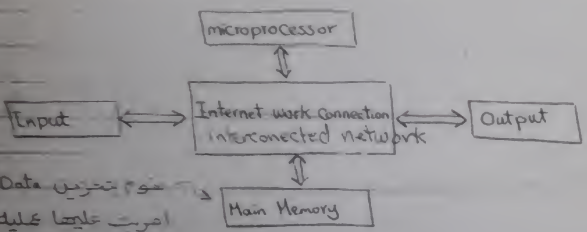
Ch 1 Microprocessor (Introduction)



الوحيد القادر على ادخال عناوين



Data و Programs عن تخزين
اضربت عليها علبة الطابعة

Control processing Unit (CPU) ≡ Machine

→ CPU components

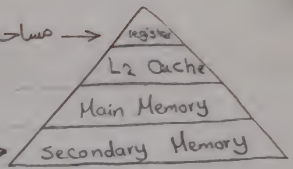
1. ALU
2. CU (Control Unit) تتحكم في وظائف الجهاز
3. Registers level 1 cache

→ CPU functions:

1. Decode arithmatic & logic instruction
   (Fetch - decode - execode)

→ Manage the program Flow (make decisions)

* Zero * negative * overflow ...



→ Transfer Data between itself and memory

processor

الشغال → Multi-processor ↔ Multi-Core

Buses: group of Common wires connect between parts of device.

1 Address Bus : →

K-bit ⟶ 2^K devices.

2 Data Bus : ↔

devices ↔ MP

← كلما زاد كلما زاد معدل نقل البيانات من وإلى الـ MP

3 Control Bus : ↔

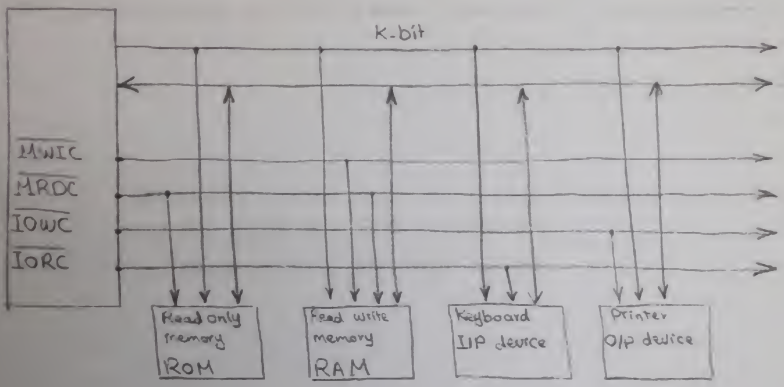
devices ↔ MP

← ينقل Control Signal

كلما زاد كلما كان أفضل

←

Draw the block diagram of a computer System showing the address data and Control bus Signal ?



* Simple Control Signal

MWIc	memory write Control Signal
MRDC	memory read Control Signal
IOWC	I/O write Control Signal
IORC	I/O read Control Signal.

→ MP → device التعامل بين

1- MP ⇝ Send address

2- Control Signal MRDC

3- Data transfer.

The Memory System is divided into "3 main points"

1- Transient program area (TPH)
2- System Area
3- extended memory System (xms)

II

→ holds
→ The length of TPA is 640 KB
→ programs → Interrupt vector

Ch 2 : The Microprocessor & Its Architecture

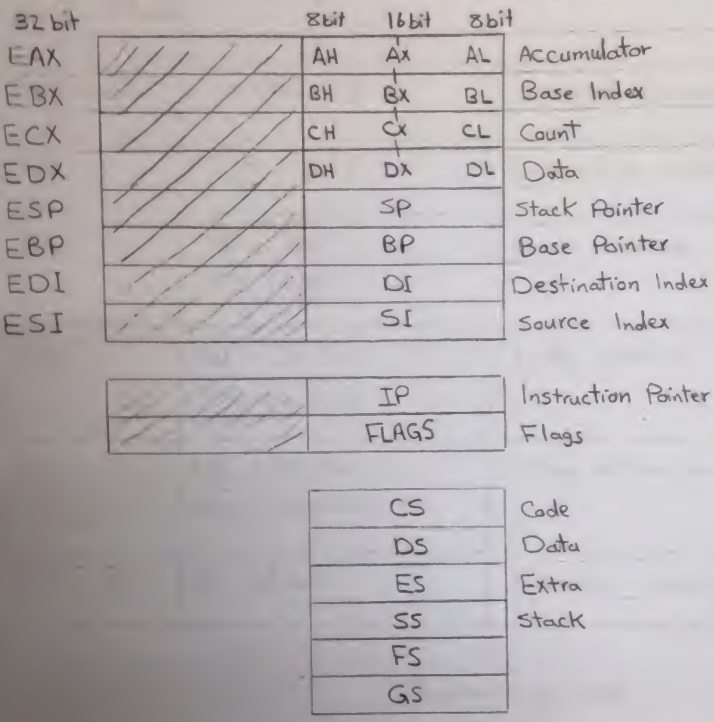
8 bit	16 bit	8 bit	
AH	AX	AL	Accumulator
BH	BX	BL	Base Index
CH	CX	CL	Count
DH	DX	DL	Data
	SP		Stack Pointer
	BP		Base Pointer
	DI		Destination Index
	SI		source Index

IP	
FLAG	

CS	Code Segement
DS	Data Segement
ES	Extra Segement
SS	Stack Segement
FS	
GS	

special purpose.

Programming Model Of The Intel 8086 Pentium 4



The Programming Model Of The Intel 80386 Pentium 4

كل ← 4 bit في الشاشة يعطي 1 bit في السادس عشر

20 bit ⟶ (XXXXX) Hexa.

Real Mode

Segement	Off Set	Special Purpose
CS	IP - EIP	Instruction address
SS	① Then ② SP or BP , ESP / EBP	Stack address
DS	BX . DI . SI 8 bit 16 bit } numbers	Data address
ES	DI , EDI For string Instructions	String destination address
FS _ GS	No default	General address.

Programming Model

program Visible
- They are used during app-prog
- Ex: EAx, Ax, AI, EBX ...
→ Mov Ax, Bx

program invisible
- not addressed directly during app-prog
- only 80286 - P4 Contain this type
- ex: FIAG, EFIAG
SUB Ax, BX بعد تنفيذ أمر

Purpose Of Use

General purpose
EAX , EBX
Hold Data address or off set +
ابعد عن البدايه

Special purpose
CS, DS, ES, SS

int X = 3
address ↑ data ↑

Addressing modes (How the memory is accessed)
Addressing mode ← كثيره لكن طرق تنظيم البيانات داخل الذاكره
وظيفتها الوصول إلى الذاكره

1- Real mode 1 MB only
2- Protected mode upto 4GB

« Real mode »
→ Accumulator EAX . AX . AL . AH
Can be used as 32bit → EAX , 16 → Ax , 8bit → AH, AL

1 Special purpose :
← طرق أساسى فى عمليات الضرب والقسمة

→ Div Ax , Bx : $\frac{Ax}{Bx}$

Ax : يوضع به الجزء الصحيح
Dx : (Data register) يوضع به باقى القسمه "الكسر"

→ Mux Ax^{16} , Bx^{16}
16 * 16 ⟶ 32 bit

→ Mux EAX , EBX
32 , 32

Dx	Ax
high bytes	low bytes

EDX	EAX
high bytes	low bytes

Base Index

Can be used as 32 bit → EBX , 16 bit → BX

8 bit → BH , BL

General purpose → hold off set for Data Segement

Note : In Real Mode

We have 64 KB processor , 5 bit memory

→ To access the memory الوصول للذاكرة

← processor من الـ

$64\,K = 2^6 \cdot 2^{10} = 12^{16}$

$(16\ bit)_8 \rightarrow (4\ bit)_{Hexa}$.

~> From 4 bit processor how to reach 5 bit memory

we add zero from the left side to Base

→ To reach to any address in the memory we need Base and off set

البداية Base is in Code Segement

$$\text{Current address}_{(real)} = (CS)_o + \text{off set}$$

Ex: CS = 4000 Hex , IP = 2342

Solution

Current address = $(cs)_o + IP = 40000 + 2342 = 42342$ Hex

Count

To Count in any process with loop we need big register

loop Cx , ECX

Repetition Cx

Shift CL we need small number of loops

Data
Can be used as 32 bit → EDX 16 bit → DX
8 bit → DH . DL
General purpose → Div , Mul العمليات الحسابية

Stack pointer
Can be used as 16 bit → SP
hold the offset of stack segement (SS) ← المؤشر الى

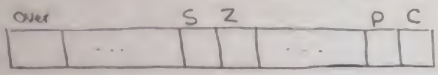
Base pointer
Can be used as 16 bit → BP
used as offset for stack segement (ss)

Destination Index
Can be used as 16 bit → DI المؤشر الثانوي بعد BX
hold the off set of Data segement

Source Index
Can be used as 16 bit → SI المؤشر الثانوي بعد DI
hold the off set of Data segement

Instruction Pointer
Can be used as 16 bit → IP
hold the offset of current instruct to be executed (in Code segement)

Flag Invisible



Can be used as 16 bit - 32 bit

→ Help microprocessor to take the corect way

1 over Flow happened when → +ve + +ve = -ve
→ -ve + -ve = +ve

2 Carry (C)

11111010

Carry 11101110+
① ← 11101000

3 Parity (P)

1 → number of one's is even

0 → number of one's is odd

تستفيد من هذا عندما يتم نقل Data من مكان لآخر نتأكد من أن Data تم نقلها بدون فقد حيث اذا كان عدد 1 زوجي بعد النقل وكذلك P=1 فان النقل تم بالصوره المطلوبه والعكس يكون فيه فقد في Data

Ex: SS = 3245 H
BP = 2342

Sol

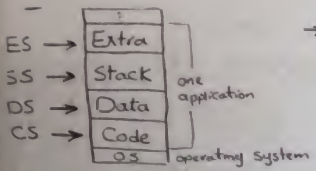
Base address = 33450 H
Current address = 33450 + 2342 = 35792
Ending address = Starting + FFFF
= 33450 + FFFF

Protected Mode

Processor $\xrightarrow[\text{2 modes}]{\text{access}}$ Memory

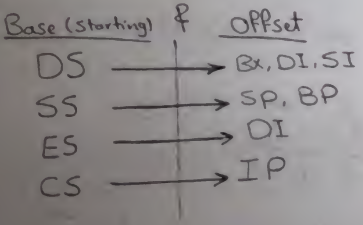
1 Real Mode

- access 1st MB memory
- run Only one application
- DOS



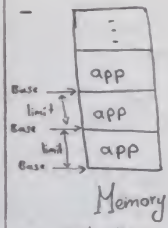
→ How to access memory?

- EA = (Base)$_0$ + offset



2 Protected Mode. (80286 and above)

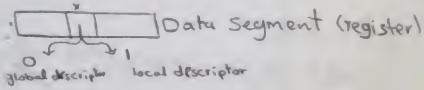
- access up to 4GB
- run more one application.
- GUT windows, linux



→ every application has different size from the other

→ How to access memory?

1- Selector: Choose one of descriptors



2- Descriptor

- each descriptor 8 byte.
- There are 8192 descriptor for every application
- → local Descriptors: describe local applications
  → Global Descriptors: describe global applications.
- Knows us ① Base: application start in memory
  ② limit: the depth of the end from base
  → end = Base + limit
  ③ access right → a- Code
  b-data → Read only / Read / write
  c-perivity

Application

- **Global (System) Application**
  - The data of this type is public.
- **Local (User) Application**
  - The data of this type is private.

Descriptors

Global Descriptor	Local Descriptor
— describe Global applications	— describe local applications.
— 8192 descriptor (number)	— 8192 descriptor (number)
— each descriptor 8 byte (size of each one in memory)	— each descriptor 8 byte (size of each one in memory)
— Total size $= \frac{8 \text{ byte} * 8192}{1024} = 64 \text{ KB}$	— Total size $= \frac{8 \text{ byte} * 8192}{1024} = 64 \text{ KB}$
— each descriptor describe • Base • limit • access right	— each descriptor describe • Base • limit • access right

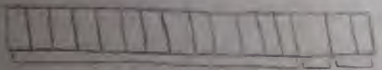
Processor عند الـ Register يصل اي الذاكرة ←

protected mode و Real mode بين الاختلاف هو

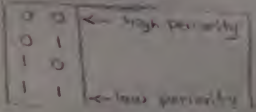
و بالتالي تكون الاختلاف بين

في طريقة تفسير ما داخل Registers حيث في Real mode

و في protected mode كالتالي و $EA = (Base)_{16} + offset$



① Perority bits



← If the operation works with any program with 00 perority and got an interrupt from another program with 01 perority the processor will still work with the first program but if the 2nd program with 11 perority and got an interrupt the

So We Know ① the type of descriptor (global - local)
② the number of descriptor (one of 0 to 8191)
③ Periority bits

From Register

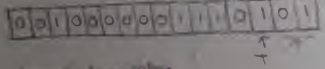
So We Can Know ① Base ② limit ③ Access right

From Descriptor ← which we get from register.

السؤال: أين تقع جداول descriptor ؟

Example:

CS = 2075 H



T=1 local descriptor

① لوحة processor

② لوحة Memory: كيف وإذا أنا أخدم الوصول الى الذاكرة عندي ان هنا access الذاكرة موجب

③ Cash Memory: ذاكرة بين processor و memory لتسريع عمل processor ومساعدتها عشان ترجع البيانات المكررة ولكنها هنا هي الجزء المتكرر

→ Cash Memory → level 1 : in processor

→ level 2 : between memory and processor

The Descriptor Formats For 80286 P4 microprocessor

7	00000000	00000000	6
5	Access rights	Base (B23 – B16)	4
3	Base (B15 – B0)		2
1	limit (L15 – L0)		0

The Descriptor Formats For 80386/80486/P/Ppro/PII microp-

7	Base (B31 – B24)	G	D	0	A V	limit (L19 – L16)	6
5	Access rights	Base (B23 – B16)					4
3	Base (B15 – B0)						2
1	limit (L15 – L0)						0

Ch3 Addressing modes (Real Mode)

		Examples
[1] Immediate Addressing Mode	I → R (I: information, R: register)	Mov BL, 44 , Mov BH, 44H Mov CL, 10011110 B Mov ECx, FC23F34EH
[2] Register Addressing Mode.	R ↔ R (register ↔ register)	Mov CL, BL (✓) Mov Cx, Bx (✓) | Mov CS, Bx (x) Mov EAx, EBx (✓) Mov Ds, Cx (✓) | Mov Bx, CS (✓) Mov Ds, CS (x)
[3] Direct Addressing Mode Direct addressing ↓ R: EAx, Ax, AH, AL Displacement (off set) addressing → Any Register.	R ↔ M (memory)	Direct: Mov AL, Num / Mov Ax, Num / Mov Num, Ax / Mov EAx, Num Displacement: Mov CL, Num / Mov Cx, Num / Mov EBx, Num / Mov Dx, [1023]

[4] Register Indirect Addressing Mode[]
↳ (BP , Bx , SI , DI)
	R ↔ [M]	Mov Ax, [Bx]
		Mov [BP], DL
		Mov EDx, [DI]
		Mov [Bx], [SP] (X)

[5] Bass_plus_Index Addressable Mode EBx → Bx, EBP → BP, ESI → SI, EDI → DI use with one dimention array	R ↔ M	Mov Dx, [Bx+DI] Mov WORD / DWORD PTR [Bx+DI], Dx

[6] Register_Relative Addressable Mode
Bx, BP, SI, DI / EBP, EDx, ESI, EDI / off set / مكانك الاراي
	R ↔ M	Mov Ax, [Bx+1000]
		Mov Ax, 1000[DI]
		Mov WORD / DWORD PTR [Bx+1000], Ax

[7] Base_relative plus Index Addressable Mode Used in two dimention Array	R ↔ M	Mov WORD PTR 1000[Bx+DI], Mov DH, [Bx+DI+20H]

Effective Address (EA)

No E.A

Notes

( )$_{10}$ عشرى

( )$_{B}$ ثنائى Binary

( )$_{H}$ السادس عشر

Characters

R ↔ R has the same size (8 ↔ 8, 16 ↔ 16)

← لا يجوز بأى حال من الأحوال تغيير Code segment (CS) الذى يحتوى على عنوان الأوامر داخل الذاكرة وكذلك لا يجوز النقل منه segement الى آخر

R_{source} (R_S)

الذى يحصل منه على البيانات Register

E.A = memory = Num

→ EA هو المكان الذى نضع به أو نأخذ منه البيانات ولا يتأثر بنوع Instruction أو بعنى امح لا يتأثر بذاته

1→ Memory ~ Byte addressable

Byte by Byte متوقعه

2→

little endian assignement

$EA = (seg)_{10} + OFF\ set$

① $EA = (DS)_0 + BX$

② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$

$EA = (DS)_0 + BX + DI$

Starting address

$EA = (DS)_0 + BX + 1000$

$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$

- عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable

Mov [BX], AL ← يتم تحديد مساحه التخزين

يتم التعديل الى ←

Mov Byte PTR [BX], AL

Mov WORD PTR [BX], AL

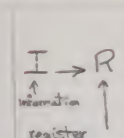
Mov DWORD PTR [BX], AL

يظل effective address ثابت

عند نقل بيانات من Memory الى Register لا تحدد المساحه للتخزين وقد توجد مشاكل

Ch 3 : Addressing modes (Real Mode)

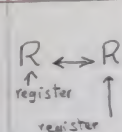
[1] Immediate Addressing Mode



Examples

Mov BL, 44 , Mov BH, 44H
Mov CL, 1001110 B
Mov ECx, FC23F34EH

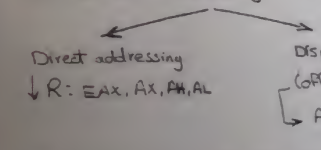
[2] Register Addressing Mode.

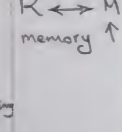


Mov CL, BL (✓)
Mov Cx, BX (✓)
Mov EAx, EBX (✓)
Mov Ds, Cx (✗)
Mov Ds, CS (✗)

Mov CS, BX (✗)
Mov BX, CS (✓)

[3] Direct Addressing Mode





Direct
Mov AL, Num
Mov Ax, Num
Mov Num, Ax
Mov EAX, Num

Displacement
Mov CL, Num
Mov Cx, Num
Mov EBx, Num
Mov Dx, [1023]

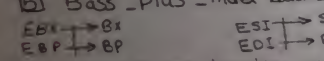
[4] Register Indirect Addressing Mode[]

↳ (BP , Bx , SI , DI)

R ↔ [H]

Mov Ax, [Bx]
Mov [BP], DL
Mov EDx, [DI]
Mov [Bx], [SP] (X)

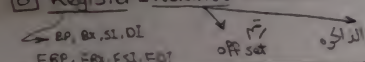
[5] Bass_Plus_Index Addressable Mode



R ↔ M

Mov Dx, [Bx+DI]
Mov WORD / DWORD PTR [Bx+DI], Dx

[6] Register_Relative Addressable_Mode



R ↔ M

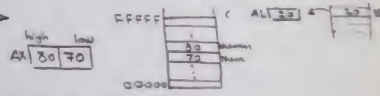
Mov Ax, [Bx+1000]
Mov Ax, 1000[DI]
Mov WORD / DWORD PTR [Bx+1000], Ax

[7] Base_relative plus Index Addressable Mode

Used in Two dimention Array

R ↔ M

Mov WORD PTR 1000[Bx+DI], A
Mov DH, [Bx+DI+20H]

Effective Address (EA)	Notes
No EA	( )$_{10}$ عشرى ( )$_B$ Binary ثنائى ( )$_H$ السادس عشر . ——— , 4 Characters
R_{source} (R_s) الذى تحصل منه على البيانات Register	$R \leftrightarrow R$ has the same size 8 ↔ 8, 16 ↔ 16 ← لا يجوز بأى حال من الأحوال تغير Code segment (CS) الذى يحتوى على عنوان أول أمر داخل الذاكره ولذلك لا يجوز النقل من segement الى آخر
E.A = memory = Num ⇒ EA هو المكان الذى نضع به أو تأخذ منه البيانات ولا يأثر بنوع Instruction أو بمعنى أصح لا يأثر بحجمه	1→ Memory ~ Byte addressable Byte by Byte مترقمه 2→  little endian assignement
$EA = (seg)_{+0} + OFFset$ ① $EA = (DS)_0 + BX$ ② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$	← عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable Mov [BX], AL ← لم يتم تحديد مساحه التخزين لذا يتم التعديل الى ← Mov Byte PTR [BX], AL Mov WORD PTR [BX], AL Mov DWORD PTR [BX], AL ويظل effective address ثابت
$EA = (DS)_0 + BX + DI$ Starting address	
$EA = (DS)_0 + BX + 1000$	
$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$	← عند نقل بيانات من Register الى Memory لا تحدد المساحه للتخزين ولا توجد مشاكل

Ch 4 : Data Movement Instructions

1- Assembler & Deassembler

2- Stack (push, pop)

3- Instructions

Assembler & Deassembler

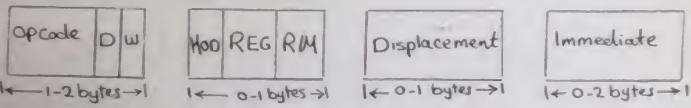
Hint : Assembler : transform the assembly Code to machine Code

Deassembler : transform the machine Code to assembly Code

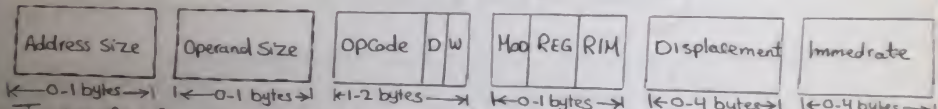
→ machine Code (0's , 1's)

↝ Modes of Instructon

[1] 16 bit Instruction mode (8086)



[2] 32-bit Instruction mode (80386 above)



To transfere from assembler to deassembler follow this steps

[1]

Operand Size	Address Size	mode
66 H X	67 H X	- 16-bit instruction mode with its default (8 - 16 bit) registers w=0 w=1
✓	X	- 16 bit instruction but works with (8 - 32 bit) registers. w=0 w=1
✓	✓	- 32 bit instruction works with its default (8 - 32 bit) register w=0 w=1
X	✓	- 32 bit instruction but works with (8 - 16 bit registers. w=0 w=1

[2] | Opcode | D | W |

→ Opcode: has machine code of the instruction (Mov, sto, swap)
we get this code from the table
like Mov ⟶ 100010

→ D ⟶ D=1 (when we transfer from Memory to register)
⟶ D=0 (when we transfer from register to Memory)

→ W ⟶ W=1 (word / Dword)
⟶ W=0 (Byte)

[3] | MOD | REG | R/M |

→ MOD : Its machine code can be get from this table

MOD	Function		Ex:
00	No Displacement	R/M is memory	Mov AL, [DI]
01	8-bit sign-extended displacement		Mov AL, [DI+2]
10	16-bit displacement		Mov AL, [DI + 1000_H]
11	R/M is a register		

→ REG : Its machine code can be get from this table

Code	W=0 (Byte)	W=1 (Word)	W=1 (Dword)
0 0 0	AL	AX	EAX
0 0 1	CL	CX	ECX
0 1 0	DL	DX	EDX
0 1 1	BL	BX	EBX
1 0 0	AH	SP	ESP
1 0 1	CH	BP	EBP
1 1 0	DH	SI	ESI
	BH	DI	EDI

Code	Seg
0 0 0	ES
0 0 1	CS
0 1 0	SS
0 1 1	DS
1 0 0	FS
1 0 1	GS

→ R/M → to get the code of R/M → first we need to know if R/M is Memory or Register that depend on MOD

IF MOD = 11 ∴ R/M is registers
we can get its code from register table

ode	W=0 (Byte)	W=1 (word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

IF MOD = (00, 01, 10) ∴ R/M is Memory
we can get its code
From this table

R/M Code	Addressing mode	Function
000	DS : [BX + SI]	DS : [EAX]
001	DS : [BX + DI]	DS : [ECX]
010	SS : [BP + SI]	DS : [EDX]
011	SS : [BP + DI]	DS : [EBX]
100	DS : [SI]	Uses scaled-index byte
101	DS : [DI]	SS : [EBP]*
110	SS : [BP]* special addressing mode	DS : [ESI]
111	DS : [BP]	DS : [EDI]
	16-bit R/M memory addressing mode	32-bit addressing mode selected by R/M

④ Displacement

① 1 byte □□ ⟶ Put as it use 2's

ex: 1 → 01 , 21 → 21

② 2 byte □□ □□ ⟶ Swap high, low

ex: 1234 → 3412

ex: 123 → 2301

Stack
(pop & push)

→ The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (last In First Out) stack memory

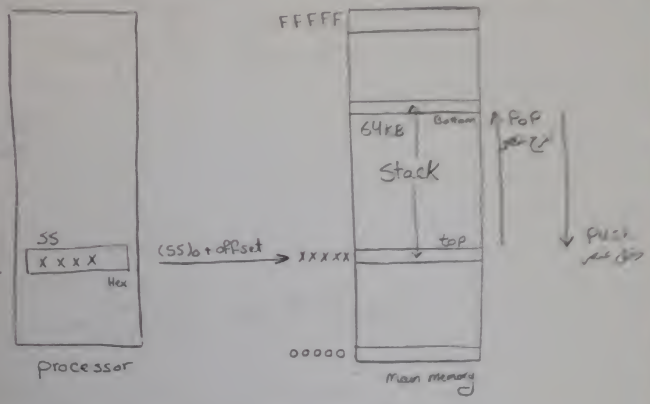
Stack (LIFO)

↓

two operations

Pop — يجيب عنصر ويحرك pointer

Push — يحرك Pointer ويدخل عنصر

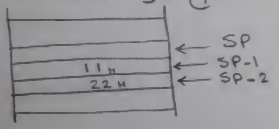


عندما Stack بالعناصر ووضع عنصر آخر فيها يحدث Cyclic nature اي يتم وضع ما بالعنوان الاول ووضع العنصر الجديد في مكانه

→ Push AX (✓)

AX: 11H (high byte) | 22H (low byte)

Stack في AX محتوى وضع



→ Push 5262H (✓)

→ Push [BX] (X) ← هنا لم يدر المساحة التي يخزن بها البيانات في memory

EA = (DS)o + BX

The Correct

Push Byte PTR [BX] (✓)

Push (word - Dword) PTR [BX] (✓)

Hint

22H can be Put in memory as

1 Byte: 22H

2 word: 00 22

3 Dword: 00 00 22

→ Push CS (✓) → هنا لم يغير محتوى CS

... performed on bytes, words or doublewords, ...

... to ... instruction ...

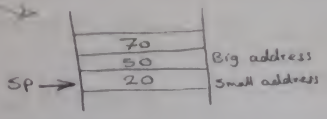
Pop ← stack ← data

Pop 2345 H (X)

Pop CS (X) → we mustn't change CS

Pop BX (✓) →

1) BX | 50 | 20 |
high low

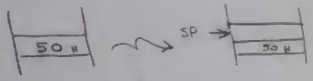


high address → high byte
low address → low byte

2) Then SP = SP + 2

Pop Byte PTR [BX]

EA = (DS)$_0$ + BX →



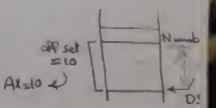
Instructions

- LEA
  - LDS, LSS, LFS, LGS, LES
  - * OFF set
- LODS, STOS, MOVS, INS, OUTS.
  - * IN, Out
  - * X ChG
  - * BSwap
  - * CMOS

→ 1) LEA (load effective address)

ex: LEA AX, Numb ≡ Mov Ax, offset Numb

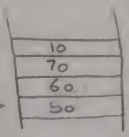
هذا الأمر معناه وضع بعد المكان Numb عن البداية في AX



ex: LEA BX, [DI] →→ EA = (DS)$_0$ + DI ≠ MOV BX, offset [DI] (X)

→ LDS

ex: LDS BX, [DI]



EA = (DS)₀ + DI = XXXXX ⟶

BX	60	50
DS	10	70

← يتم تحديد بدايه العنوان من EA

← نأخذ من memory بيانات تحجم حجم المكان التى سوف توضع به

Hinit

byte = 2 bit

memory size

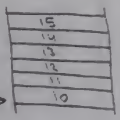
في المثال BX تحجم 16 bit اى 2 byte ← نأخذ مكانين من الذاكره ونضعهم في BX بحيث يوضع العنوان الاول من الذاكره في low byte والمكان الثانى في high byte

← ثم نأخذ العنوانين التاليين ونضعهم في DS مثل آن الأمر LDS

→ LSS

ex: lss EAX, [BX + DI]

EA = (DS)₀ + BX + DI = XXXX →



EAX	13	12	11	10

where EAX → 32 bit

SS	15	14

→ LES

ex: lES BX, [DI]

EA = (DS)₀ + DI = XXXXX ⟶

BX	60	50

, ES | 80 | 70 |

→ LFS

ex: LFS BX, [DI]

EA = (DS)₀ + DI = XXXXX ⟶

BX	60	80

, FS | 70 | 50 |

→ LGS

ex: LGS BX, [DI]

EA = (DS)₀ + DI = XXXXX →

BX	8	1

, GS	4	3

LCS EAX, [DI] (X)

because CS mustn't be Changed.

String Instructrions

LODS, STOS, MOVS, INS, OUTS

This Instructions depend on direction"C

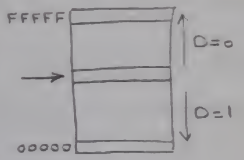
	D		FLAG

D=0 ⟶ autoincrement mode

تسير في اتجاه زياده العنوان

D=1 ⟶ autodecrement mode

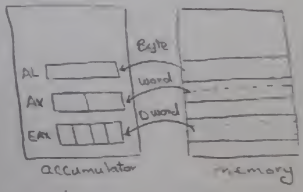
تسير في اتجاه نقص العنوان



[1] LODS

accumulator في وضع memory من نأخذ

ثم نزود أو نقلل حسب offset D



LODSB ⟶ byte

LODSW ⟶ word

LODSD ⟶ double word

المكان غير معروف مباشره بوصله EA عن طريق

LODS list ⟶ byte (DB)

LODS Data ⟶ word (DW)

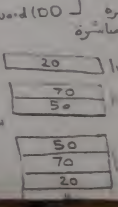
LODS FROG ⟶ double word (DD)

المكان الذي تم وضع فيه في الذاكره مدروس مباشره

list: DB 20 H

list2: DW 5070 H

list3: DD 50702011 H



and Can be Signed integer (IMUL) or Unsigned

From [Source] → to [Destination]

[Data Segment] — offset SI

[Extra Segment] — DI

ex: DS = 1000H , SI = 1000H , D = 0
execute LODSW then LODSD

Solution

EA = $(DS)_0$ + SI = 10000 + 1000 = 11000

$SI_{new} = SI_{old} + 2\,bit = 1002$

AX [90 | 50]

11002 →	90
11000 →	50

D = 0

To execute LODSD

SI = 1002 , DS = 1000 , D = 0

EA = 10000 + 1002 = 11002

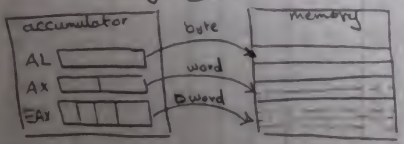
EAX [7 | 6 | 5 | 10]

$SI_{new} = SI_{old} + 4\,bit = 1002 + 4 = 1006$

If D=1 $SI_{new} = SI_{old} -$ (1 → byte, 2 → word, 4 → Dword)

[2] STOS

memory في accumulator يضع



D يزود أو يقلل ← offset

STOSB , STOSW , STOSD
STOS list (DB) , STOS list2 (DW)
STOS list2 (DD)

[3] MOVZX

Register يزود باقي خانات بـ أصفار Zeros

MOVSX (Memory) يزود باقي المساحة بنفس القيمة في الإشارة

[4] XCHG

→ Both places must have the same SIZ

→ CS mustn't be Change

Examples:

XCHG AL, CL	(✓)
XCHG CX, BP	(✓)
XCHG Data, AX	(✓)
XCHG DS, ES	(X)
XCHG CS, AX	(X)

[5] BSwap

→ work only with 32 bit registers
(80486 → P3, P4)

EAX [50 | 20 | 30 | 40] before

EAX [40 | 30 | 20 | 50] after

[6] CMOV (Conditional Mov)

نقل مشروط

CMOVZ → هذا الأمر معناه
1- يرى خانة Flag is Z
Z = 0 لا يفعل شيء
Z = 1 يفعل

ex: CMOVZ AX, BX

If Z=1 ∴ AX = BX

else Do nothing

→ CMOVC → Chick Carry

If C=1 → Move

else Do nothing

→ CMOVNC → Chick No Carry

If C=0 → Move

else Do nothing

Ch 5 Arithematic & Logic Instruction

[1] Arithematic

- ADD
- ADC
- SUB
- SBB
- INC
- DEC
- MUL
- IMUL
- DIV
- IDIV
- CMP
- JA, JB, JAE
- JBE, JE, JNE

(ADD, ADC, SUB, SBB: binary operator; INC, DEC: binary operator)

[2] Logic

- AND
- OR
- NOT
- NEGATE
- TEST
- XOR
- BT
- BTC
- BTS
- BTR
- Shift → (logical) unsigned numbers → SHL (*2), SHR (/2)
- Shift → (Arithematic) signed numbers → SAL, SAR
- Rotate → Without Carry → ROR, ROL
- Rotate → With Carry → RCR, RCL

→ Arithematic

① ADD

→ ADD AL, 20H → $AL_{new} = AL_{old} + 20$ [Immediate Addressing mode] EA = No E.A

→ ADD AL, BH → $AL_{new} = AL_{old} + BH$ [Register Addressing mode] E.A = BH

→ ADD AL, NUM → $AL_{new} = AL_{old} + NUM$ [Direct Addressing mode] EA = NUM

→ ADD AL, [BX] → $AL_{new} = AL_{old}$ + place in memory (1 byte) [Register Indirect Addressing Mode]

EA of [BX] = $(DS)_0 + BX$ ⟶ ⊢⊣

→ ADD Ax, [Bx + 2000] → $Ax_{new} = Ax_{old}$ + place in memory (2 byte) [Register - Relative Addressing Mode]

$EA = (DS)_0 + BX + 2000$

→ ADD EAx, [Bp + SI] → $EAx_{new} = EAx_{old}$ + place in memory (4 byte) [Bass plus Index Addressing Mode]

$EA = (SS)_0 + BP + SI$

→ ADD Ax, 2000 [Bx + SI] → $Ax_{new} = Ax_{old}$ + place in memory (2 byte) [Base relative plus Index A.M]

$EA = (DS)_0 + BX + SI + 2000$

→ ADD 2000[BX+SI], EAX ⟶ place in memory (4 byte) = place in memory (4 byte) + EAX

EA = $(DS)_0 + BX + SI + 2000$

V.I

① ADD ES, DS (✓)

② ADD [BX], [BP] (✗)

③ ADD CS, AX (✗)

④ ADD AX, CS (✓)

⑤ ADD CX, BL (✗)

③ CMP (Compare) → we Compare between them to make ...

② ADC (add with Carry)

Carry → C

Bx Bx +

Bx Cx

Dx Ax

→ ADC AL, 20H → $AL_{new} = AL_{old} + 20_H + C$

وبأخذ باقي أشكال Addressing Modes مثل الأمثلة السابقة لـ ADD

③ SUB

→ SUB AL, 20H → $AL_{new} = AL_{old} - 20_H$

وأيضا هو وباقي Instructions تأخذ نفس أشكال Addressing Modes مثل الأمثلة السابقة لـ ADD

④ SBB (Sub with Borrow) Borrow : Flag في Carry خانة هي نفس

→ SBB AL, 20H → $AL_{new} = AL_{old} - 20_H - C$

⑤ INC (Add 1)

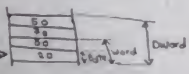
Operators

Unary: INC, DEC — Binary: ADD, SUB — Ternaly: X > 3 ? Yes: y=10, NO: y=18

→ INC AX → $AX_{new} = AX_{old} + 1$

V.I → INC Byte ptr / word ptr / Dword ptr [BX]

$EA = (DS)_0 + BX$



→ Byte 20 + 1 = 21

→ word 80 20 + 1 = 80 21

→ Dword 50 70 80 20 + 1 = 50 70 80 21

→ INC ES (X)

→ INC CS (X)

⑥ DEC (Sub 1)

→ DEC AX → $AX_{new} = AX_{old} - 1$

→ DEC ES (X)

→ DEC CS (X)

Multiplication is performed on bytes, words or doublewords, and can be Signed integer (IMUL) or unsigned integer (MUL). The product after a multiplication is always a double-width product.

Object / type	Examples:
(a) 8 bit Multiplication – The multiplicand is always in the AL register – The multiplier can be 8-bit register or any memory location – After the multiplication, the unsigned product is placed in AX a double-width product	MUL BL MUL CL MUL Temp
(b) 16 bit Multiplication – The multiplicand is always in the AX register – The multiplier can be 16 bit register or any memory location – After the multiplication the unsigned product is placed in EAX a double-width product.	MUL CX MUL word ptr [SI]
(c) 32 bit Multiplication – Only the 80386 through P4 processors multiply 32 bit double word – The multiplicand is always in the EAX register – The multiplier can be 32 bit register or any memory location – The product (64 bit mode) is found in EDX-EAX where EAX contains the least-significant 32 bit of the product	MUL ECX MUL Dword ptr [ECX]
(a) 8 bit Multiplication – AL is multiplied by a bit register (ex. DH) or any memory location (ex. [BX]), the signed product is in AX	IMUL DH IMUL byte ptr [BX]
(b) 16 bit Multiplication – AX is multiplied by 16 bit register (ex. DI), – The signed product is in DX - AX	IMUL DI
(c) 32 bit Multiplication – EAX is multiplied by 32 bit register (ex. EDI) – The signed product is in EDX-EAX → Only the 80386 P4 processors multiply 32 bit double word.	IMUL EDI

(iii) CMP

Division occurs on 8- or 16 bit numbers in the 8086 - 80286 proc
and on 32 bit numbers in the 80386 P4. These numbers
signed (IDIV) or unsigned (DIV) integers The dwidend is alway
a double-width dividend that is divided by the operand.
There is no immediate division instruction any micro

	Object / type	Example
(i) DIV The dividend is used to store the dividend.	ⓐ 8-bit Division – AX register to store the dividend – The dividend is divided by the contents of any 8 bit register or memory location. – The quotient moves into AL after the division with AH containing a whole number remainder.	DIV CL DIV Byte ptr [BP]
	ⓑ 16-bit Division – is similar to 8 bit division except that instead of dividing into Ax. The 16 bit number is divided into DX - AX, a 32 bit dividend	DIV Cx DIV NUMB
	ⓒ 32 bit Division – The 64 bit contents of EDX - EAX are divided by the operand specified by the instruction. leaving a 32-bit quotient in EAX and a 32 bit remainder in EDX	DIV ECX DIV DATA2
(ii) IDIV	ⓐ 8-bit Division – Ax is divided by 8 bit register (ex BL), the signed quotient is in AL and the remainder is in AH.	IDIV BL
	ⓑ 16-bit Division – DX-Ax is divided by 16-bit register (ex SI), the signed quotient is in Ax and the remainder is in DX	IDIV SI
	ⓒ 32 bit Division – EDX EAX is divided by the doubleword contents of the data segment memory location addressed by EDI, the signed quotient is in EAX and the remainder is in EAX	IDIV DWORD PTR [E

⑪ CMP (Compare)

→ We Compare between things to make some operations depend on the relation (< , = , >) between this things.

→ CMP AL, BH

① AL − BH
- AL > BH → Result +ve Then S=0 , Z=0
- AL = BH → Result 0 Then Z=1
- AL < BH → Result −ve Then S=1 , Z=0

Note: This Operation Changes S and Z in Flag

JA → A>B
JB → B>A
JAE → A≥B
JBE → B≥A
JE → A=B
JNE → A≠B

→ CMP AL , BH
JA Lable1
JB Lable2
other instructions
Lable1 : ...
Lable2 : ...

① AL − BH
a) Result +ve ⇝ AL > BH
- Jumb to Lable 1 to do its operations neglecting JB Lable2 and the other instructions.
b) Result −ve ⇝ BH > AL
- Jumb to Lable 2 to do its operations, neglecting the other instructions.
c) If AL isnot > BH or BH > AL
- go to do other instructions.

→ CMP [AL], [BH] (X)
→ CMP CS, SS (X)
→ CMP CS, Ax (✓)
→ CMP Ax, AL (X)

★ ═══════ ★

→ Logic

① AND

→ AND AL, 20H

assume AL = 10H

AL = 00010000
20H = 00100000 and
AL new = 00000000 = 0H

② OR

→ OR AL, 20H

AL = 10H

AL = 00010000
20H = 00100000 OR
AL new = 00110000 = 30H

③ NOT (1's complement)

→ NOT Byte ptr / word ptr / Dword ptr [BX]

1's Complement → 0 ⇝ 1 , 1 ⇝ 0

④ NEGATE (2's complement)

Note: 2's Complement

تمشي من اليمين تنزل 0 وأول واحد 1 يقابلنا تنزله وبعده تقلب 0 اى 1 والعكس

ex: 111010 , 001
2's Complement: 000110 , 111

★ S ★

④ TEST

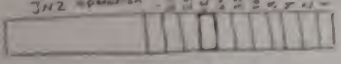
يختبر الوزن الموضعي (1,1,0) الخانه حدد رقمها
وبالتالي بليه امر آخر JZ , JNZ

→ TEST Ax ,128

JZ operation
JNZ operation

JZ
↓
Jump if 0

JNZ
↓
Jump if no Zero

Ax  ← result ?

* ——— *

⑤ BT

يختبر bit واحده الموجود فقط في (80386-P4)

→ BT AX , 4

Test bit position 4 in Ax

* ——— *

⑥ BTC → test and complement

→ BTC Ax , 128
JZ op1
JNZ op2

← يقوم بعمل Test لـ bit واحده
ثم [1] اذا كانت بـ 0 يحولها الى 1
[2] اذا كانت بـ 1 يحولها الى 0

* ——— *

⑦ BTS → Test and Set

→ BTS Ax, 4

Test and set bit position 4 in Ax

← يقوم بعمل Test لـ bit واحده
ثم يضع لها 1

* ——— *

⑧ BTR → Test and Reset

→ BTR Ax, 4

Test and Reset bit position 4 in Ax

← يقوم بعمل Test لـ bit واحده
ثم يضع لها 0

(II) Shift : Shift instructions position or move numbers to the left or right within a register or memory location.

[1] Unsigned (logical) Numbers

→ Shift Right SHR

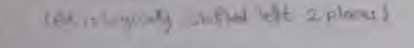
Ex: SHR, BX, 12 (BX is logically shifted right 12 places)

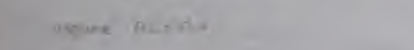
SHR ECX, 10 (ECX is logically shifted right 10 places)

→ Shift Left SHL

Ex: SHL AX, 2 (AX is logically shifted left 2 places)

SHL AL, 2 assume AL = 11





[2] Signed (Arithmatic) numbers

→ Shift Right SAR

← في هذا النوع لابد من الحفاظ على الإشارة

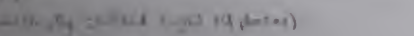
Ex: SAR SI, 2

(SI is arithmatically shifted right 2 places)

SAR EDX, 14

(EDX is arithmatically shifted right 14 places)

→ Shift Left SAL

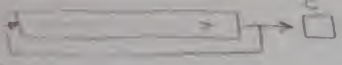


SAL AL, 2 assume AL = 22H

the information in a register or memory location, either from one end to another or through the Carry Flag.

[1] Rotate without Carry

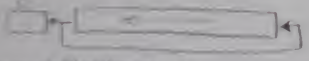
→ Rotate Right ROR



Ex: ROR WORD PTR [BP], 2

(the word contents of the stack segment memory location addressed by BP rotate right 2 places)

→ Rotate Left ROL



Ex: ROL SI, 14 (SI rotates left 14 places)

ROL ECX, 18 (ECX rotates left 18 places)

[2] Rotate With Carry

→ Rotate Right RCR

Ex: RCR AH, CL (AH rotates right through carry the number of places specified in CL)

→ Rotate Left RCL

Ex: RCL BL, 6 (BL rotates left through carry 6 places)